



PROGRAMMING Expert

There's a lot more to PassKey than password protection. This month Mike James perfects his PassKey program, which enables users to store website logins and passwords easily and securely

PROGRAMMING NEWS

WORLD CUP FOR PROGRAMMERS

The Imagine Cup programming contest is in its fourth year with \$125,000 (around £70,000) of prizes on offer for students who can "imagine a world where technology enables us to live healthier lives". The competition pits teams of students from around the world against each other in six 'invitationals' on various aspects of programming. The global final will take place in Delhi in August 2006. Visit www.imaginecup.co.uk for details.

DESKTOP SUPERCOMPUTING HAS ARRIVED

Wolfram Research's Mathematica Personal Grid Edition uses the parallel-processing techniques of supercomputers to allow today's multiple-core PCs to solve maths problems faster. It combines the standard facilities in Mathematica with parallelism and is designed to run on affordable quad-core machines. If you have a computationally intense task, visit www.wolfram.com/personalgrid.

THE HOME ROBOT IS BORN

New Japanese robotics company ZMP claims to have built the first personal humanoid robot, the nuvo. Designed by Ken Okuyama, the creative director at Pininfarina in Turin, whose major works include the Ferrari Enzo and Ferrari Rosa, the nuvo is undeniably stylish. It's 39cm in height, weighs 2.5kg and responds to voice commands or instructions issued through a web browser. When prompted, nuvo will trot up and extend its hand, announce the time and date and play music programmed by its owner. Its most practical function is taking photos, giving it a role as a monitor or security device. You can buy one from www.dynamism.com/nuvo.

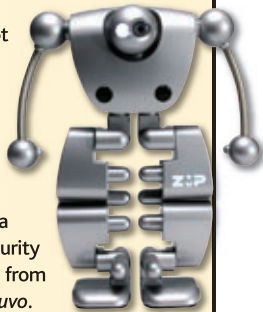
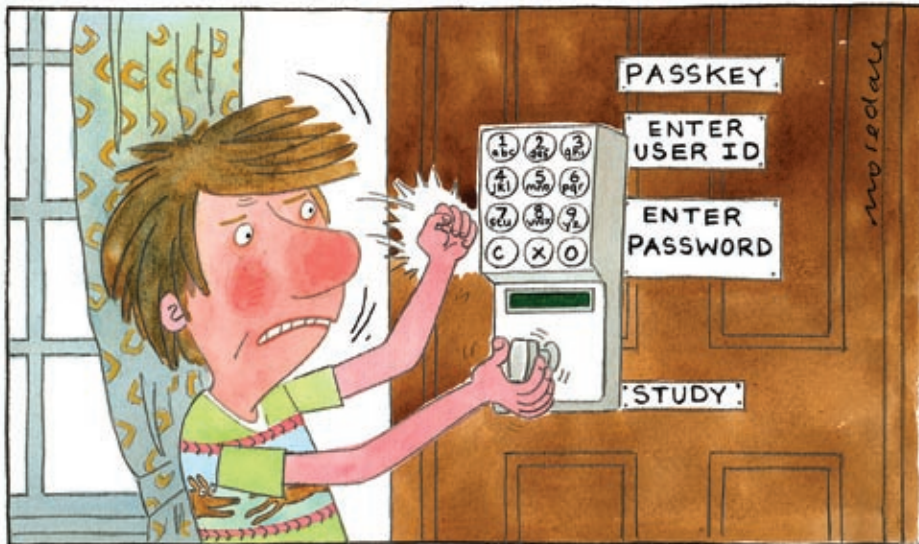


ILLUSTRATION: MIKE MOSEDALE



Last month, we developed the PassKey program for storing usernames and passwords and entering them easily into websites. The finished program stored logon details and other information in an encrypted form and could enter them into web pages with a mouse click – you'll find the full article at www.computershopper.co.uk under 'Last issue'. This month, we are going to refine PassKey to make it more practical and useful. If you don't fancy typing in the code, you can get this from your cover disc too.

SPACE SAVER

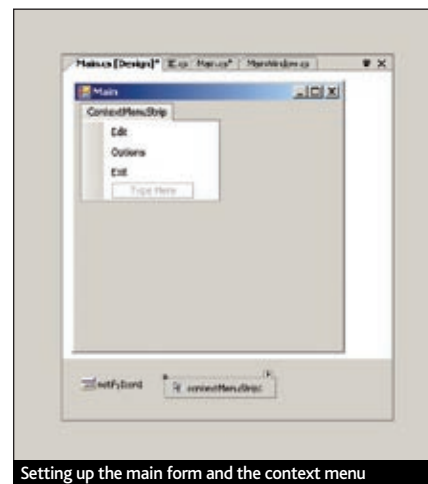
One of the problems with last month's version of PassKey is that it takes up too much space. Ideally you want it to pop up quickly, take you to a website, log you in and then go away until you need it again. Windows provides the System Tray for pop-up tools such as this. The System Tray is the small area to the right of the taskbar, and here you will find icons that represent a range of utilities such as MSN Messenger.

Usually you can right-click a System Tray icon and display a context menu from which you can select commands. Clicking or double-clicking on the icon brings up the main application window. This low-profile approach is ideal for PassKey, and the .NET languages make it fairly easy to implement. The trick is to use a NotifyIcon and a ContextMenu control on a form. These two controls implement a System Tray application for you, but there is one small problem. The easiest way to create these controls is to put them on a form and let the form designer deal with initialising them and so on. However, you don't want that form ever to appear on the screen. But you do want it to handle any events the controls generate. The solution is to

add a new Main form just to host the components and handle the events.

Load last month's project from the cover CD. Add a new form and call it Main. Drag and drop a NotifyIcon and a ContextMenu control from the toolbox to the form. To make the two controls work together, you need to set the NotifyIcon's ContextMenuStrip property to the name of the ContextMenu that you just placed on the form.

You also have to assign an Icon to its Icon property because otherwise nothing will appear in the System Tray when you run it. If you are using C# 2005 Express, the bad news is that it doesn't come with an icon editor, but there are a few available for download as freeware or shareware. Alternatively, you can convert a small bitmap to an icon file at www.chami.com/html-kit/services/favicon.



Setting up the main form and the context menu



Setting up the right-click context menu couldn't be easier. If you select the ContextMenu control, a menu appears that you can edit directly. In this case, you type in the three menu items Edit, Options and Exit. Double-clicking on any of the menu items also generates a click event handler for it.

We need this new form to act as the startup form for the project. To do this, open the file Program.cs using the Solution Explorer and change the main method to read:

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.Run(new Main());
}
```

All applications start this way, using the Run method to create the main form.

Returning to the Main form, we have to code the event handlers for each of the menu items and deal with any initialisation required. As we don't want the Main form to appear on the screen, we must modify its Constructor to:

```
public Main()
{
    InitializeComponent();
    TopLevel = false;
}
```

Setting a form's TopLevel property to false means that it is owned by a parent form. However, if the form's Parent property is null, the form simply vanishes and the system never calls its Paint method, hence it becomes invisible.

THE MAINWINDOW

Once the form has loaded we use its Load event handler to display the icon in the System Tray and to create an instance of the application's MainWindow. The MainWindow is simply our original form renamed – the one that does all the work through the DataGridView control:

```
private void Main_Load(object sender,
    EventArgs e)
{
    notifyIcon1.Visible = true;
    MW = new MainWindow();
    MW.Visible = false;
}
```

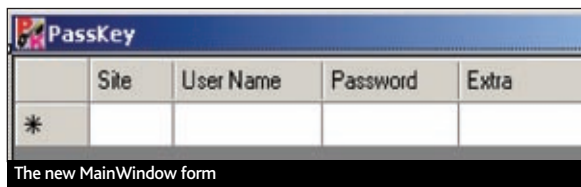
We also need to declare the MW variable so that we can keep track of the MainWindow:

```
private MainWindow MW;
```

When the user double-clicks the System Tray icon, we need to display the MainWindow. This is accomplished by:

```
private void notifyIcon1_DoubleClick(
    object sender, EventArgs e)
{
    MW.Visible = true;
    MW.Show();
}
```

If you run the program at this stage, you will see the icon appear in the System Tray. If you right-click



the icon the context menu will appear, and if you double-click it the main window will appear. As yet, the context menu doesn't do anything. We need to modify the MainWindow form.

EXTRA HELP

The first version of the program used a form to do all the work, but here we'll build most of the user interface into the right-click context menu. This means we can remove the two buttons and set the grid to fill the entire form. Select 'Dock in parent container'. It would also be useful if we entered an extra security item at some websites. To accommodate this, simply add an extra column called Extra. There is no need to see the URL of the website when using PassKey, so set that column's Visible property to False. The final version of the MainWindow form should look like the screenshot above.

Next we need to edit some of the existing code. The first big change is that all the code that was in the form's Load event handler must now be placed in the constructor – which, with the benefit of hindsight, is where we should have put it in the first place. It can't stay where it is, because now the form isn't loaded until the user displays it, but the password and data have to be available before this.

We want to allow the user to dismiss the form by clicking the close box, but we don't want the form to unload itself. All that's necessary is for the form to hide itself, so we need to supply a FormClosing event handler:

```
private void MainWindow_FormClosing(
    object sender, FormClosingEvent
    Args e)
{
    if (e.CloseReason ==
        CloseReason.UserClosing)
    {
        this.Visible = false;
        e.Cancel = true;
    }
}
```

This sets the form's status to invisible and then cancels the close operation. To bring the whole application to an end, we need to code the Exit item on the context menu. In the new Main form we need to add:

```
private void ExitToolStripMenuItem_
    Click_1(
    object sender, EventArgs e)
{
    Application.Exit();
}
```

Now if we run the program, we can display the MainWindow by double-clicking, hide it again by clicking the close box and then terminate the entire application by selecting Exit from the right-click menu.

THE SHOWEDITMODE

Now that we have got rid of the Edit button on the MainWindow, we need another way of allowing the user to edit and enter data. The simplest way is to add a ShowEditMode method to the MainWindow and call it when the user clicks the Edit item of the pop-up menu:

```
private void editToolStripMenuItem_
    Click(
    object sender, EventArgs e)
{
    MW.ShowEditMode();
}
```

The ShowEditMode method has to convert the grid into a form that allows editing, and show it modally so that it waits until the user has finished editing before restoring it to non-edit mode. We must change the form title, make the URL column visible again and display the grid:

```
public void ShowEditMode()
{
    Boolean temp = this.Visible;
    Visible = false;
    string tempstring = this.Text;
    Text = "Close to finish editing";
    dataGridView1.Columns[1].Visible =
        true;
    editing = true;
    dataGridView1.EditMode =
        DataGridViewEditMode.EditOnEnter;
    this.ShowDialog();
}
```

When the user has finished editing, we undo these changes:

```
editing = false;
dataGridView1.Columns[1].Visible =
    false;
Visible = temp;
Text = tempstring;
dataGridView1.EndEdit();
dataGridView1.Refresh();
WriteData();
}
```

To make sure the data file is updated, we then write the contents of the grid out using the WriteData method.

CONTROLLING IE

In the first version of the program, we started up Internet Explorer when the user clicked on the name of a website in column zero of the grid. When they subsequently clicked on data in the grid, it was sent to that instance of IE. This approach has several problems. The most obvious is that the user can start multiple copies of IE and each new instance replaces the previous one, which is then lost as far as PassKey's control is concerned. A better approach is to restrict the user to working with one instance of IE at a time and to manage that instance so that PassKey detects when the user closes it and allows them to create another instance to work with a different website. You could take the alternative approach of allowing the user to work with multiple instances of IE at the same time – this isn't difficult.

To control the instance of IE that PassKey is working with, the best approach is to wrap the



REVIEW BOOK

Visual Basic
2005 Jumpstart:
Pocket Guide

RATING ★★

PRICE £7

SUPPLIER www.amazon.co.uk

ISBN 059610071X

PAGES 214

Jumpstart is a nice image: you can imagine author Wei-Meng Lee getting out his jump leads for instant action. But in practice the start is not quite as smooth as you might wish.

The book is a bit of a mess, introducing ideas in a fairly random order. If you are struggling with some of the difficult ideas involved in switching from VB6, which is a partially object-oriented language, to VB 2005, which is almost 100 per



cent object-oriented, this book might be the last straw. It covers far too much ground in its 200 pages and throws in 'helpful' hints and tips that often have nothing to do with the main problems of using VB 2005. The examples are too long and too ambitious, and they don't really help you sort out the problems or understand the ideas. In many ways, the code is more about impressing the reader with what VB 2005 can do than aiding understanding.

This is more like a book of magic tricks than an aid to understanding VB 2005. You might like it if you have a short attention span, but it's unlikely to teach you much about writing VB 2005 programs.



Short and cheap



A book with no idea what the beginner needs to know

REVIEW BOOK

Microsoft Visual
C# 2005 Express
Edition: Build a
Program Now!

RATING ★★★★★

PRICE £9

SUPPLIER www.amazon.co.uk

ISBN 0735622299

PAGES 200



gradually through creating an application.

The problem is that the application quickly evolves into something a bit too sophisticated for the beginner who's struggling with more basic ideas. The author seems to be under the misapprehension that the purpose of an example is to show what's possible rather than how things work. This book is simply in too much of a hurry to get somewhere, and as a result it doesn't take the time to explain fundamental principles. You could easily work your way through this book thinking you were making progress only to discover that when cast adrift you had no idea what to do next.

It's not a good introduction to programming – still, it does come with some useful software.



Colourful, exciting and comes with a copy of C# Express



Many of the examples are too difficult to illustrate a point

instance in a new object that keeps track of it. Add a new class, called IE, to the project. We need some global variables:

```
private Int32 ProcessHandle;
private Process IEproc;
private const string IElocation =
    @"C:\Program Files (x86)\
    Internet Explorer\IEXPLORE.EXE";
```

Set the IElocation constant to the location of IEXPLORE.EXE on your machine. We are also using a range of class libraries, so add the following to the initial 'using' statements:

```
using Microsoft.VisualBasic;
using System.Threading;
using System.Windows.Forms;
using System.Diagnostics;
using System.Reflection;
```

The constructor does the work of getting the instance of IE running and loading the required URL:

```
public IE(string URL)
{
    ProcessHandle = Interaction.Shell(
        IElocation + " " + URL,
        AppWinStyle.NormalFocus, false,
        0);
    IEproc = Process.GetProcessById(
        ProcessHandle);
    IEproc.EnableRaisingEvents =
        true;
    IEproc.Exited +=
        new EventHandler(IEproc_
            Exited);
}
```

Last month, we used the same Shell method to start IE running. However, this month, we'll also use a Process class to keep

track of it. The Shell method returns a process number, which you can use to create an associated process object, IEproc, that wraps the original process. After creating the IEproc process object, it is set so it can raise events and define an event handler. Usually the designer creates and manages event handlers, but in this case we have to do the job ourselves.

The Exited event is raised when the process wrapped by IEproc ends. We can use this to detect the fact that the user has closed the web browser. Apart from the Exited event handler, which we will return to in a moment, the only method we require is SendText:

```
public void SendText(string Text)
{
    if (IEproc.Responding)
    {
        Interaction.AppActivate(Process
            Handle);
        Thread.Sleep(100);
        SendKeys.SendWait(Text);
    }
}
```

We use the IEproc object to check that the instance of IE is still responding to user input before sending the keystrokes to it.

DECLARING AN EVENT DELEGATE

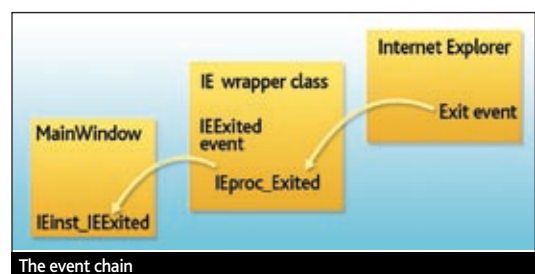
Finally we need to code the event handler. Ideally we need to make changes to the MainWindow when the instance of IE closes, so the best thing to do is to pass the event on rather than do anything with it. This is very simple in principle but, as you will see, there is a hidden problem that makes the task more difficult.

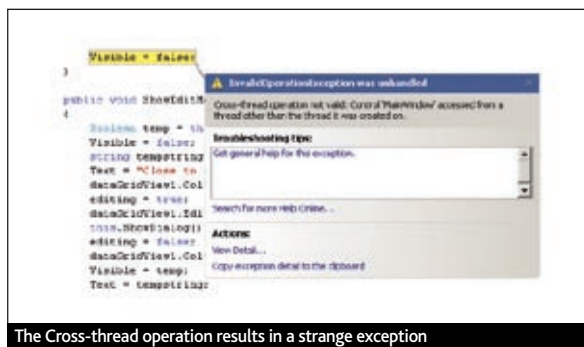
Let's take the simple route first. To pass on an event, we need to declare a public event delegate. You can think of delegates as variables that store a list of functions or methods. This is how events work: each time you set up an event handler, it is added to the delegate that's used when the event is raised. Normally this involves defining a suitable delegate first, but in this instance we just want to pass on an existing event so we might as well use the existing delegate, which is called EventHandler.

Add the following as a global variable:

```
public event EventHandler IEEExited;
```

This gives IE a public event called IEEExited, which other classes can use to attach event handlers of their own in the form of delegates. For this to work, however, we must also implement a routine to raise the event by calling all the delegates associated with the event. Put simply, raising the event is just a matter of calling the functions that have been assigned to the IEEExited delegate, and we can do this in the IEproc_Exited event handler:





The Cross-thread operation results in a strange exception

```
void IEproc_Exitd(
    object sender, EventArgs e)
{
    if (IEExited != null)
    {
        IEExited(this, e);
    }
}
```

This really couldn't be simpler. When the Exited event occurs, it calls IEproc_Exitd, and this in turn calls any methods that have been assigned to the delegate that stores the IEExited delegate list.

CLASS MANAGEMENT

Now that we have implemented this, we can go back to the MainWindow class and implement a better DataGridView click event handler. This is very similar to the original, but it uses the new IE class. If the user clicks in column zero, the program checks to see if an instance of IE exists. If not, it creates one and an event handler is provided for the IEExited event:

```
private void dataGridView1_CellClick(
    object sender, DataGridViewCell
    EventArgs e)
{
    if (editing) return;
    if (e.ColumnIndex == 0)
    {
        if (IEinst == null)
        {
            ActiveRow = e.RowIndex;
            IEinst = new IE(
                (String)dataGridView1.Rows
                [ActiveRow].
                Cells[e.ColumnIndex +
                1].Value);
            IEinst.IEExited +=
                new EventHandler(IEinst_
                IEExited);
        }
    }
}
```

If the user clicks on another column then as long as an instance of IE exists, the contents of the corresponding row and column are sent to it:

```
if (e.ColumnIndex > 1)
{
    if (IEinst != null)
    {
        IEinst.SendText(
            (String)dataGridView1.
            Rows[ActiveRow].
            Cells[e.ColumnIndex].Value);
    }
}
```

```
}
}
```

The event handler is also simple:

```
void IEinst_IEExited(
    object sender,
    EventArgs e)
{
    IEinst = null;
    ActiveRow = 0;
    Visible = false;
}
```

When the instance of IE is closed, its wrapper IEinst is disposed of and the MainWindow vanishes from the screen.

CROSS-THREADING

If you try the program as given, it seems to work until you close the IE window and an unhandled exception occurs: "Cross-thread operation not valid: Control "MainWindow" accessed from a thread other than the thread it was created on."

This might come as a surprise for two reasons. First of all, we haven't explicitly created another thread of execution, and second, this sort of technique used to work under .NET 1.1. Most of the Forms class library isn't threaded, but until .NET 2.0 there was nothing to stop you creating potentially dangerous programs by allowing multiple threads to work with forms or components. The question is what to do about it. What we need to do is arrange for the thread that created the form to call the form's event handler. This should be easy in principle, but in practice it is difficult to get right.

If you look for examples of how to call a routine on the thread that 'owns' it, you will see some simple code, but it assumes you know which object is being called in this way at compile time. In this case, raising the event could result in the calling of a set of methods in various objects that aren't determined until run time. In the jargon, the event handlers are 'late bound'. To deal with this, we must use the .NET reflection library, which enables you to investigate what an object is and call its methods dynamically. We start in the same general way:

```
void IEproc_Exitd(
    object sender, EventArgs e)
{
    if (IEExited != null)
    {

```

We must use the form's Invoke method to run the event handler using the thread that created the form. Invoke(delegate, parameters) will run the method specified in the delegate and pass it the specified parameters as an array of objects, using the correct thread of execution. To get the correct version of the form's Invoke method, we specify its signature (the types of the parameters it accepts) as a Type array.

```
Type[] paramTypes = new Type[2];
paramTypes[0] = typeof(Delegate);
paramTypes[1] = typeof(object[]);
```

We also need to create an object array that contains the parameters we want to pass to the form's event handler:

```
object[] param = new object[2];
param[0] = this;
param[1] = e;
```

However, we can't pass this directly to the event handler because the event handler will be called by the form's Invoke method, and its name and parameters have to be specified as an object array:

```
object[] Dinvoke = new object[2];
Dinvoke[1] = param;
```

The Dinvoke[0] element will be set to the delegate that will be called to handle the event. The event delegate IEExited contains an array of delegates that have been added to it as event handlers. Each one has to be called to handle the event. We need the array of delegates:

```
Delegate[] List =
    IEExited.GetInvocationList();
```

Next we use a for loop to call each delegate in turn:

```
MethodInfo MInfo;
Type targettype;
for (int i = 0; i < List.Length;
    i++)
{

```

We set the first element of Dinvoke to the delegate to be called:

```
Dinvoke[0] = List[i];
```

Then we have to get the form's Invoke method:

```
targettype = List[i].Target.Get
    Type();
MInfo = targettype.GetMethod(
    "Invoke", paramTypes);
```

Now we have the form's Invoke method in the MethodInfo class. This too has an Invoke method, which you can use to call the method it represents on the instance you specify:

```
MInfo.Invoke(List[i].Target,
    Dinvoke);
}
```

At this point you may be thinking there are too many methods called 'Invoke' and too many delegates to keep track of. It took us some time to get the logic of this code correct. Admittedly, we haven't tested it with anything other than the MainWindow form that uses it. If you plan to use this approach in more complicated situations, you may well have to modify it.

GOING FURTHER

The PassKey program is incredibly useful, and there is much we could do to make it even better. The Options menu item needs to be implemented to display a dialog box that lets you save and restore backups of the encrypted file. A printing option would also be useful, as would more onscreen security, and the MainWindow could be customised further as well. What you choose to add is a matter of personal taste and the environment in which you are using it. ☐